

53

C#, Visual Basic, C++/CLI, and F#

WHAT'S IN THIS CHAPTER?

- Namespaces
- Defining types
- Methods
- Arrays
- Control statements
- Loops
- Exception handling
- Inheritance
- Resource management
- Delegates
- Events
- Generics
- LINQ queries
- C++/CLI mixing native and managed code

C# is *the* programming language designed for .NET. More than 50 languages exist for writing .NET applications — for example, Eiffel, Smalltalk, COBOL, Haskell, Pizza, Pascal, Delphi, Oberon, Prolog, and Ruby. Microsoft alone delivers the languages C#, Visual Basic, C++/CLI, J#, JScript.NET, and F#.

Every language has advantages and disadvantages; some things can be done easily with one language but are complicated with another one. The classes from the .NET Framework are always the same, but the syntax of the language abstracts various features from the framework. For example, the C# `using` statement makes it easy to use the objects implementing the `IDisposable` interface. Other languages need more code for the same functionality.

The most commonly used .NET languages from Microsoft are C# and Visual Basic. C# was newly designed for .NET with ideas from C++, Java, Pascal, and other languages. Visual Basic has its roots in Visual Basic 6 and was extended with object-oriented features for .NET.

C++/CLI is an extension to C++ that is an ECMA standard (ECMA 372). The big advantage of C++/CLI is the ability to mix native code with managed code. You can extend existing native C++ applications and add .NET functionality, and you can add .NET classes to native libraries so that they can be used from other .NET languages such as C#. It is also possible to write completely managed applications with C++/CLI.

F# is a new language within Visual Studio that offers special support for functional programming. It works well with .NET because it supports traditional object-oriented programming as well. To give you some information what functional programming with F# means: Functions can be used as values. This makes it easy to compose functions from multiple functions, and do function pipelining where functions are chained after each other.

This chapter shows you how to convert .NET applications from one language to another. If you see sample code with Visual Basic or C++/CLI, you can easily map this to C#, and the other way around. Don't assume you can learn F# or C++/CLI with this comparison because it mainly gives information about how the C# ideas map to the other languages, but not to the core concepts and ideas of the other languages.



For this chapter, I assume that you know C# and have read the first few chapters of this book. It is not necessary to know Visual Basic, C++/CLI, and F#.

NAMESPACES

.NET types are organized into namespaces. The syntax for defining and using namespaces is quite different between the four languages.

To import namespaces, C# uses the `using` keyword. C++/CLI is fully based on the C++ syntax with the `using namespace` statement. Visual Basic defines the `Imports` keyword, F# defines the `open` keyword, to import namespaces.

With C# and Visual Basic, you can define an alias to classes or other namespaces. With C++/CLI, an alias can reference other namespaces, but not classes. C++ requires the `namespace` keyword to define an alias — the same keyword is used to define a namespace. Visual Basic uses the `Imports` keyword again.

For defining namespaces, all four languages use the `namespace` keyword, but there's still a difference. With C++/CLI, you can't define hierarchical namespaces with one namespace statement; instead the namespaces must be nested. There's one important difference with the project settings: defining a namespace in the project settings of C# defines a default namespace that shows up in the code of all new items that you add to the project. With Visual Basic project settings, you define the root namespace that is used by all items in the project. Namespaces declared in the source code define only the sub-namespace inside the root namespace:

```
// C#
using System;
using System.Collections.Generic;
using Assm = Wrox.ProCSharp.Assemblies;
namespace Wrox.ProCSharp.Languages
{
}

// C++/CLI
using namespace System;
using namespace System::Collections::Generic;
namespace Assm = Wrox::ProCSharp::Assemblies;
namespace Wrox
{
    namespace ProCSharp
    {
        namespace Languages
```

```

        {
        }
    }

' Visual Basic
Imports System
Imports System.Collections.Generic
Imports Assm = Wrox.ProCSharp.Assemblies
Namespace Wrox.ProCSharp.Languages

End Namespace

// F#
namespace Wrox.ProCSharp.Languages
open System
open System.Collections.Generic

```

DEFINING TYPES

.NET differentiates between reference types and value types. With C#, reference types are defined with classes, and value types with structs. In addition to reference and value types, this section also shows you how to define an interface (a reference type) and an enumeration (a value type).

Reference Types

To declare a reference type, C# and Visual Basic use the `class` keyword. In C++/CLI, a class and a struct are nearly the same; you don't have the separation between a reference type and a value type as you do with C# and Visual Basic. C++/CLI has a `ref` keyword to define a managed class. You can create a reference type by defining `ref class` or `ref struct`.

Both with C# and C++/CLI the class is surrounded by curly brackets. With C++/CLI, don't forget the semicolon at the end of the class declaration. Visual Basic uses the `End Class` statement at the end of the class. F# creates classes and all other types with the `type` keyword followed by the name of the type and the optional `as` identifier, which defines the name of the instance identifier:

```

// C#
public class MyClass
{
}

// C++/CLI
public ref class MyClass
{
};
public ref struct MyClass2
{
};

' Visual Basic
Public Class MyClass2
End Class

// F# signature file
type MyClass() as this =
// ... members of the class

```

When using a reference type, a variable needs to be declared, and the object must be allocated on the managed heap. When declaring a handle to a reference type, C++/CLI defines the handle operator `^`, which is somewhat similar to the C++ pointer `*`. The `gcnew` operator allocates the memory on the managed heap. With C++/CLI,

it is also possible to declare a variable locally, but for reference types, the object is still allocated on the managed heap. With Visual Basic, the variable declaration starts with the statement `Dim` followed by the name of the variable. With `new` and the object type, memory is allocated on the managed heap:

```
// C#
MyClass obj = new MyClass();
var obj2 = new MyClass()

// C++/CLI
MyClass^ obj = gcnew MyClass();
MyClass obj2;

' Visual Basic
Dim obj as New MyClass2()
```

If a reference type does not reference memory, different keywords are used: C# defines the `null` literal, C++/CLI defines `nullptr` (`NULL` is valid only for native objects), and Visual Basic defines `Nothing`. With F#, the `null` value is not normally used — it's not a value permitted with types that are defined with F#. By using types defined by languages other than F#, you might get `null` values.

Predefined reference types are listed in the following table. C++/CLI does not define the `object` and `string` type as is done with the other languages. Of course, you can use the classes defined by the framework.

.NET TYPE	C#	C++/CLI	VISUAL BASIC	F#
System.Object	object	Not defined	Object	obj
System.String	string	Not defined	String	string

Value Types

To declare a value type, C# uses the `struct` keyword; C++/CLI uses the keyword `value`; Visual Basic, Structure, and F#, use the `type` keyword with the attribute `[<Struct>]`. An alternative is to use the `struct` and `end` keywords:

```
// C#
public struct MyStruct
{
}

// C++/CLI
public value class MyStruct
{
};

' Visual Basic
Public Structure MyStruct
End Structure

// F#
[<Struct>]
type MyStruct =
    val x : int

type MyStruct2 =
    struct
        val x : int
    end
```

With C++/CLI, you can allocate a value type on the stack, on the native heap by using the `new` operator, and on the managed heap by using the `gcnew` operator. C# and Visual Basic do not have these options, but they become important when native and managed code is mixed with C++/CLI:

```
// C#
MyStruct ms;

// C++/CLI
MyStruct ms1;
MyStruct* pms2 = new MyStruct();
MyStruct^ hms3 = gcnew MyStruct();

' Visual Basic
Dim ms as MyStruct
```

Predefined value types for the different languages are listed in the following table. In C++/CLI, the `char` type has a size of just 1 byte for an ASCII character. In C#, `char` has a size of 2 bytes for Unicode characters; that's a `wchar_t` in C++/CLI. The ANSI standard for C++ just defines `short <= int <= long`. With 32-bit machines, `int` and `long` both have a size of 32 bits. To define a 64-bit variable in C++, you need `long long`.

.NET TYPE	C#	C++/CLI	VISUAL BASIC	F#	SIZE
Char	char	wchar_t	Char	char	2 bytes
Boolean	bool	bool	Boolean	bool	1 byte, contains true or false
Int16	short	short	Short	int16	2 bytes
UInt16	ushort	unsigned short	UShort	uint16	2 bytes with no sign
Int32	int	int	Integer	int	4 bytes
UInt32	uint	unsigned int	UInteger	uint	4 bytes with no sign
Int64	long	long long	Long	int64	8 bytes
UInt64	ulong	unsigned long long	ULong	uint64	8 bytes with no sign

Type Inference

C# allows you to define a local variable without an explicit data type declaration with the `var` keyword. The type is inferred from the initial value that is assigned. Visual Basic offers the same feature using the `Dim` keyword as long as *Option infer* is turned on. This can be done with the compiler option `infer+` or by using the project configuration page with Visual Studio. With F#, the type of values, variables, parameters, and return types is inferred:

```
// C#
var x = 3;

' Visual Basic
Dim x = 3

// F#
let x = 3
```

Interfaces

Defining interfaces is very similar for the three languages that use the keyword `interface`. It's different with F#, because interfaces are defined with a `type` that contains only abstract members:


Available for
download on
Wrox.com

```
// C#
public interface IDisplay
{
    void Display();
}
```

code snippet CSbarp/Person.cs

Available for
download on
Wrox.com

```
// C++/CLI
public interface class IDisplay
{
    void Display();
};
```

code snippet CPPCLI/Person.hAvailable for
download on
Wrox.com

```
' Visual Basic
Public Interface IDisplay
    Sub Display
End Interface
```

code snippet VisualBasic/Person.vbAvailable for
download on
Wrox.com

```
// F#
[<Interface>]
type public IDisplay
    abstract member Display : unit -> unit
```

code snippet FSharp/Person.fs

Implementing interfaces is different. C# and C++/CLI use a colon after the class name followed by the interface name. The methods defined with the interface are implemented. With C++/CLI, the methods must be declared `virtual`. Visual Basic uses the `Implements` keyword to implement an interface, and the methods that are defined by the interface also need the `Implements` keyword attached. F# lists the implemented interfaces in the section within the type definition. The interface is implemented with the interface and with keywords and implements all members of the following interface:

Available for
download on
Wrox.com

```
// C#
public class Person: IDisplay
{
    public void Display()
    {
    }
}
```

code snippet CSharp/Person.csAvailable for
download on
Wrox.com

```
// C# explicit interface implementation
public class Person: IDisplay
{
    void IDisplay.Display()
    {
    }
}

// C++/CLI
public ref class Person: IDisplay
{
public:
    virtual void Display() { }
};
```

code snippet CPPCLI/Person.hAvailable for
download on
Wrox.com

```
' Visual Basic
Public Class Person
    Implements IDisplay
    Public Sub Display Implements IDisplay.Display
    End Sub
End Class
```

code snippet VisualBasic/Person.vb



```
// F#
type Person as this =
    interface IDisplay with
        member this.Display() = printfn "%s %s" this.FirstName this.LastName
```

code snippet FSharp/Person.fs

Enumerations

Enumerations are defined similarly in three languages with the `enum` keyword (only Visual Basic uses a new line instead of a comma to separate the elements). F# uses the `type` keyword and the `match |` expression:



```
// C#
public enum Color
{
    Red, Green, Blue
}
```

code snippet CSharp/Color.cs



```
// C++/CLI
public enum class Color
{
    Red, Green, Blue
};
```

code snippet CPPCLI/Color.h



```
' Visual Basic
Public Enum Color
    Red
    Green
    Blue
End Enum
```

code snippet VisualBasic/Color.vb



```
// F#
type Color =
| Red = 0
| Green = 1
| Blue = 2
```

code snippet FSharp/Color.fs

METHODS

With the exception of F#, methods are always declared within a class. The syntax from C++/CLI is very similar to C# except that the access modifier is not part of the method declaration but is written before that. The access modifier must end with a colon. With Visual Basic, the `Sub` keyword is used to define a method. With F#, a function can be defined with the `let` keyword. `Foo` has one parameter, `x`, and returns `x` plus 3. The parameter types and the return type can be declared with a function, as shown. With the `add` function, `x` and `y` are of type `int`, and an `int` is returned. If the types are not declared, the types are inferred by the compiler. With the `Foo` function, `x` is `int` as well because 3 is added to `x`, and 3 is an `int`. Defining a method within a class is done with the `member` keyword. Instance methods have a prefix that is defined with the `as` identifier in the class definition. F# does not necessarily use `this`, `Me`, or `self` to access the current instance, but you can define any identifier you like:

```
// C#
public class MyClass
{
    public void Foo()
```

```

    {
    }
}

// C++/CLI
public ref class MyClass
{
public:
    void Foo()
    {
    }
};

' Visual Basic
Public Class MyClass1
    Public Sub Foo
    End Sub
End Class

// F# function
let foo x = x + 3
let add (x : int, y : int) : int = x + y

// method within a class
type MyClass as this =
    member this.Foo() =
        printfn "MyClass.Foo"

```

Method Parameters and Return Types

With C# and C++/CLI, parameters that are passed to methods are defined inside a bracket. The type of the parameter is declared before the variable name. If a value is returned from a method, the method is defined with the type to return instead of `void`.

Visual Basic uses `Sub` statements to declare a method without returning a value, and the `Function` statement with a method that does have a return type. The return type is followed after the method name and the brackets. Visual Basic also has a different order with variable declaration and type in the parameter. The type follows the variable, which is the reverse direction from C# and C++/CLI.

With F#, if nothing is returned from a method, it is declared a `unit`. `unit` is similar to C# `void`. The method `Foo` is declared to receive an `int` parameter and returns an `int`:

```

// C#
public class MyClass
{
    public int Foo(int i)
    {
        return 2 * i;
    }
}

// C++/CLI
public ref class MyClass
{
public:
    int Foo(int i)
    {
        return 2 * i;
    }
};

```



```

' Visual Basic
Public Class MyClass2
    Public Sub Foo1(ByVal i as Integer)
    End Sub
    Public Function Foo(ByVal i As Integer) As Integer
        Return 2 * i
    End Sub
End Class

// F#
type MyClass as this =
    member this.Foo(i : int) : int = i * 2

```

Parameter Modifiers

By default, value types are passed by value, and reference types are passed by reference. If a value type that is passed as a parameter should be changed within a calling method, with C# you can use the parameter modifier `ref`.

C++/CLI defines a managed reference operator `%`. This operator is similar to the C++ reference operator `&` except that `%` can be used with managed types and the garbage collector can keep track of these objects in case they are moved within the managed heap.

With Visual Basic, the keyword `ByRef` is used for passing parameters by reference:

```

// C#
public class ParameterPassing
{
    public void ChangeVal(ref int i)
    {
        i = 3;
    }
}

// C++/CLI
public ref class ParameterPassing
{
public:
    int ChangeVal(int% i)
    {
        i = 3;
    }
};

' Visual Basic
Public Class ParameterPassing
    Public Sub ChangeVal(ByRef i as Integer)
        i = 3
    End Sub
End Class

```

When invoking a method with reference parameters, only the C# language requires you to apply a parameter modifier. C++/CLI and Visual Basic don't differentiate calling a method with or without the parameter modifier. C# has the advantage here because you can immediately see in the calling method that the parameter values can be changed.

Because of the caller syntax, which is not differentiated, Visual Basic does not allow you to overload methods just by changing the modifier. The C++/CLI compiler allows you to overload the method just by changing the modifier, but you cannot compile the caller because the resolved method is ambiguous.

With C# it is possible to overload and use methods with just the parameter modifier, but it's not good programming practice:

```
// C#
ParameterPassing obj = new ParameterPassing();
int a = 1;
obj.ChangeVal(ref a);
Console.WriteLine(a);    // writes 3

// C++/CLI
ParameterPassing obj;
int a = 1;
obj.ChangeVal(a);
Console.WriteLine(a);    // writes 3

' Visual Basic
Dim obj as new ParameterPassing()
Dim i as Integer = 1
obj.ChangeVal(i)
Console.WriteLine(i)    // writes 3
```



C# also defines the `out` keyword when a parameter is just returned from a method. This option is not available from C++/CLI and Visual Basic. As long as the caller and callee are in the same application domain, there's really no difference between `out` and `ref` behind the scenes, and you can use a method declared with the C# `out` parameter modifier from Visual Basic and C++/CLI in the same way as `ref` parameter modifiers. If the method is used across application domains or processes, the attribute `[out]` can be used with Visual Basic and C++/CLI.

Constructors

With both C# and C++/CLI, the constructor has the same name as the class. Visual Basic uses a procedure named `New`. The `this` and `Me` keywords are used to access a member of this instance. When invoking another constructor within a constructor, a member initialization is required with C#. With C++/CLI and Visual Basic, it is possible to invoke the constructor as a method.

F# looks a little bit different. One constructor is defined with the type declaration. The constructor with the `Person` gets two `string` parameters. Other constructors are defined with the `new` keyword. `new()` defines a constructor without parameters. With the `Person` class, the constructor is invoked with two parameters:



```
// C#
public class Person
{
    public Person()
        : this("unknown", "unknown") { }
    public Person(string firstName, string lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    private string firstName;
    private string lastName;
}
```

code snippet CSharp/Person.cs

Available for
download on
Wrox.com

```
// C++/CLI
public ref class Person
{
public:
    Person()
    {
        Person("unknown", "unknown");
    }
    Person(String^ firstName, String^ lastName)
    {
        this->firstName = firstName;
        this->lastName = lastName;
    }
private:
    String^ firstName;
    String^ lastName;
};
```

code snippet CPPCLI/Person.hAvailable for
download on
Wrox.com

```
' Visual Basic
Public Class Person
    Public Sub New()
        Me.New("unknown", "unknown")
    End Sub
    Public Sub New(ByVal firstName As String, ByVal lastName As String)
        Me.MyFirstName = firstName
        Me.MyLastName = lastName
    End Sub
    Private MyFirstName As String
    Private MyLastName As String
End Class
```

code snippet VisualBasic/Person.vbAvailable for
download on
Wrox.com

```
// F#
type Person(firstName0 : string, lastName0 : string) as this =
    let mutable firstName, lastName = firstName0, lastName0
    new () = Person("unknown", "unknown")
```

code snippet FSharp/Person.fs

Properties

To define a property, C# requires a get and set accessor within a property block. C#, C++/CLI, and Visual Basic also define a shorthand notation where a custom implementation is not needed if just a simple variable is returned or set by the get and set accessors. This is an automatic property. The syntax is different both with C++/CLI and Visual Basic. Both of these languages have a `property` keyword, and it is necessary to define a variable `value` with the set accessor. C++/CLI also requires a return type with the get accessor and a parameter type with the set accessor.

With the short version of writing a property with C++/CLI, you have to define the type and the name of the property; the get and set accessors are created automatically by the compiler. If nothing more than setting and returning a variable is needed, the short version is good enough. If the implementation of the accessors requires more — for example, checking the value or doing a refresh — you must write the full syntax for the properties.

F# defines properties as a member of the class using `get()` for the set accessor and `set (value)` for the set accessor:



Available for
download on
Wrox.com

```
// C#
public class Person
{
    private string firstName;
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    public string LastName { get; set; }
}
```

code snippet CSharp/Person.cs



Available for
download on
Wrox.com

```
// C++/CLI
public ref class Person
{
private:
    String^ firstName;
public:
    property String^ FirstName
    {
        String^ get()
        {
            return firstName;
        }
        void set(String^ value)
        {
            firstName = value;
        }
    }
    property String^ LastName;
};
```

code snippet CPPCLI/Person.h



Available for
download on
Wrox.com

```
' Visual Basic
Public Class Person
    Private myFirstname As String
    Public Property FirstName()
        Get
            Return myFirstname
        End Get
        Set(ByVal value)
            myFirstname = value
        End Set
    End Property
    Public Property LastName As String
End Class
```

code snippet VisualBasic/Person.vb



Available for
download on
Wrox.com

```
// F#
type Person(firstName0 : string, lastName0 : string) as this =
    let mutable firstName, lastName = firstName0, lastName0
    member this.FirstName
        with get() = firstName
        and set(value) = firstName <- value
    member this.LastName
        with get() = lastName
        and set(value) = lastName <- value
```

code snippet FSharp/Person.fs

With C# and C++/CLI, read-only properties just have a get accessor. With Visual Basic, you must also specify the `ReadOnly` modifier. Write-only properties must be defined with the `WriteOnly` modifier and a set accessor:

```
' Visual Basic
Public ReadOnly Property Name()
Get
    Return myFirstName & " " & myLastName
End Get
End Property
```

Object Initializers

With C# and Visual Basic, properties can be initialized using an object initializer. The properties can be initialized using curly brackets similar to an array initializer. The syntax from C# and Visual Basic is very similar; Visual Basic just uses the `With` keyword:

```
// C#
Person p = new Person { FirstName = "Tom", LastName = "Turbo" };

' Visual Basic
Dim p As New Person With { .FirstName = "Tom", .LastName = "Turbo" }
```

Extension Methods

Extension methods are the foundation of LINQ. With both C# and Visual Basic, it is possible to create extension methods. However, the syntax is different. C# marks an extension method with the `this` keyword in the first parameter, Visual Basic marks an extension method with the attribute `<Extension>`. With F#, the term for extension methods is type extension. Type extensions are done by specifying the fully qualified name of the type with the `type` declaration and extending it with the `with` keyword:

```
// C#
public static class StringExtension
{
    public static void Foo(this string s)
    {
        Console.WriteLine("Foo {0}", s);
    }
}

' Visual Basic
Public Module StringExtension
    <Extension()> _
    Public Sub Foo(ByVal s As String)
        Console.WriteLine("Foo {0}", s)
    End Sub
End Module

// F#
module Wrox.ProCSharp.Languages.StringExtension
type System.String with
    member this.Foo = printfn "String.Foo"
```

STATIC MEMBERS

A static field is instantiated only once for all objects of the type. C# and C++/CLI both use the `static` keyword; Visual Basic offers the same functionality with the `Shared` keyword.

To use static members, you use the name of the class followed by the `.` operator and the name of the static member. C++/CLI uses the `:` operator for accessing static members:



```
// C#
public class Singleton
{
    private static SomeData data = null;
    public static SomeData GetData()
    {
        if (data == null)
        {
            data = new SomeData();
        }
        return data;
    }
}
// use:
SomeData d = Singleton.GetData();
```

code snippet CSharp/Singleton.cs



```
// C++/CLI
public ref class Singleton
{
private:
    static SomeData^ hData;
public:
    static SomeData^ GetData()
    {
        if (hData == nullptr)
        {
            hData = gcnew SomeData();
        }
        return hData;
    }
};
// use:
SomeData^ d = Singleton::GetData();
```

code snippet CPPCLI/Singleton.h



```
' Visual Basic
Public Class Singleton
    Private Shared data As SomeData
    Public Shared Function GetData() As SomeData
        If data Is Nothing Then
            data = new SomeData()
        End If
        Return data
    End Function
End Class
' Use:
Dim d as SomeData = Singleton.GetData()
```

code snippet VisualBasic/Singleton.fs

ARRAYS

Arrays are discussed in Chapter 6, “Arrays and Tuples.” The `Array` class is always behind the scenes of .NET arrays; declaring an array, the compiler creates a class that derives from the `Array` base class. When C# was designed, the designers of the C# language took the bracket syntax for arrays from C++ and extended it with array initializers:

```
// C#
int[] arr1 = new int[3] {1, 2, 3};
int[] arr2 = {1, 2, 3};
```

If you use brackets with C++/CLI, you create a native C++ array but not an array that is based on the `Array` class. To create .NET arrays, C++/CLI introduced the `array` keyword. This keyword uses a generic-like syntax with angle brackets. Within the angle brackets, the type of the elements is defined. C++/CLI supports array initializers with the same syntax as C#:

```
// C++/CLI
array<int>^ arr1 = gcnew array<int>(3) {1, 2, 3};
array<int>^ arr2 = {1, 2, 3};
```

Visual Basic uses braces for arrays. It requires the last element number instead of the number of elements with the array declaration. With every .NET language, arrays begin with element number 0. This is also the same for Visual Basic. To make that clearer, Visual Basic 9 introduced the `0 To` number expression with the array declaration. It always starts with 0; `0 To` just makes this more readable.

Visual Basic also supports array initializers if the array is initialized with the `New` operator:

```
' Visual Basic
Dim arr1(0 To 2) As Integer()
Dim arr2 As Integer() = New Integer(0 To 2) {1, 2, 3};
```

F# offers different ways to initialize an array. Small arrays can be created by using `[|` and `|]` as opening and closing braces and specifying all elements separated by semicolons. A sequence expression such as the `for` in statement can be used to initialize elements in the array as done with `arr2`. `arr3` is initialized with the `zeroCreate` type extension to the `Array` class to create and initialize 20 elements. Arrays can be accessed using an indexer. With slices it is possible to access a range from the array — `arr2.[4..]` accesses all elements starting with the fifth element to the last:

```
// F#
let arr1 = [| 1; 2; 3 |]
let arr2 = [| for i in 1..10 -> i |]
let arr3 : int array = Array.zeroCreate 20

printfn "%d" arr1.[0]
// array slice
arr2.[4..]
```

CONTROL STATEMENTS

Control statements define what code should run. C# defines the `if` and `switch` statements, and the conditional operator.

if Statement

The C# `if` statement is the same as the C++/CLI version. Visual Basic uses `If-Then/Else/End If` instead of curly brackets:

```
// C# and C++/CLI
if (a == 3)
{
    // do this
}
else
{
    // do that
}

' Visual Basic
If a = 3 Then
' do this
Else
' do that
End If
```

Conditional Operator

C# and C++/CLI support the conditional operator, a lightweight version of the `if` statement. In C++/CLI, this operator is known as a *ternary* operator. The first argument has a Boolean result. If the result is true, the first expression is evaluated; otherwise, the second one is evaluated. Visual Basic has the `IIf` function in the Visual Basic Runtime Library, which offers the same functionality. With F#, `if/then/else` is used for the conditional operator:

```
// C#
string s = a > 3 ? "one": "two";

// C++/CLI
String^ s = a > 3 ? "one": "two";

' Visual Basic
Dim s As String = IIf(a > 3, "one", "two")

// F#
if a = 3 then printfn "do this" this else printfn "do that"
```

switch Statement

The `switch` statement looks very similar in C# and C++/CLI, but there are important differences. C# supports strings with the case selection. This is not possible with C++. With C++ you have to use `if-else` instead. C++/CLI supports an implicit fall-through from one case to the next. With C#, the compiler complains if there's not a `break` or a `goto` statement. C# has only implicit fall-through if there's not a statement for the case.

Visual Basic has a `Select/Case` statement instead of `switch/case`. A `break` is not only not needed but also not possible. An implicit fall-through from one case to the next is not possible, even if there's not a single statement following `Case`; instead, `Case` can be combined with `And`, `Or`, and `To` — for example, `3 To 5`.

F# has the match expressions with the `match` and `with` keywords. Every line of the matches starts with `|` followed by a pattern. The example uses the OR pattern with `Suit.Heart | Suit.Diamond`. The last pattern is the wildcard pattern:

```
// C#
string GetColor(Suit s)
{
    string color;
    switch (s)
    {
        case Suit.Heart:
        case Suit.Diamond:
            color = "Red";
            break;
        case Suit.Spade:
        case Suit.Club:
            color = "Black";
            break;
        default:
            color = "Unknown";
            break;
    }
    return color;
}

// C++/CLI
String^ GetColor(Suit s)
{
    String^ color;
```



```

switch (s)
{
    case Suit::Heart:
    case Suit::Diamond:
        color = "Red";
        break;
    case Suit::Spade:
    case Suit::Club:
        color = "Black";
        break;
    default:
        color = "Unknown";
        break;
}
return color;
}

' Visual Basic
Function GetColor(ByVal s As Suit) As String
    Dim color As String = Nothing
    Select Case s
        Case Suit.Heart And Suit.Diamond
            color = "Red"
        Case Suit.Spade And Suit.Club
            color = "Black"
        Case Else
            color = "Unknown"
    End Select

    Return color
End Function

// F#
type SuitTest =
    static member GetColor(s : Suit) : string =
        match s with
        | Suit.Heart | Suit.Diamond -> "Red"
        | Suit.Spade | Suit.Club -> "Black"
        | _ -> "Unknown"

```

LOOPS

With loops, code is executed repeatedly until a condition is met. Loops with C# are discussed in Chapter 2, “Core C#,” including: `for`, `while`, `do..while`, and `foreach`. C# and C++/CLI are very similar with regard to looping statements; Visual Basic and F# defines different statements.

for Statement

The `for` statement is similar with C# and C++/CLI. With Visual Basic, you can’t initialize a variable inside the `for/to` statement; you must initialize the variable beforehand. `for/to` doesn’t require a `Step` to follow — `Step 1` is the default. Just in case you don’t want to increment by 1, the `Step` keyword is required with `for/to`. F# uses the `for/to/do` and `for/downto/do` keywords:

```

// C#
for (int i = 0; i < 100; i++)
{
    Console.WriteLine(i);
}

// C++/CLI

```

```

for (int i = 0; i < 100; i++)
{
    Console.WriteLine(i);
}

' Visual Basic
Dim count as Integer
For count = 0 To 99 Step 1
    Console.WriteLine(count)
Next

// F#
for i = 1 to 10 do
    printfn i
for i = 10 downto 1 do
    printfn i

```

while and do . . . while Statements

The while and do...while statements are the same in C# and C++/CLI. Visual Basic has very similar constructs with Do While/Loop and Do/Loop While. F# doesn't have a do/while but a while/do:

```

// C#
int i = 0;
while (i < 3)
{
    Console.WriteLine(i++);
}
i = 0;
do
{
    Console.WriteLine(i++);
} while (i < 3);

// C++/CLI
int i = 0;
while (i < 3)
{
    Console::WriteLine(i++);
}
i = 0;
do
{
    Console::WriteLine(i++);
} while (i < 3);

' Visual Basic
Dim num as Integer = 0
Do While (num < 3)
    Console.WriteLine(num)
    num += 1
Loop
num = 0
Do
    Console.WriteLine(num)
    num += 1
Loop While (num < 3)

// F#
let i = 0
let mutable looping = true
while looping do

```

```

printfn "%d" i
i++
if i >= 3
    looping <- false

```

foreach Statement

The `foreach` statement makes use of the interface `IEnumerable`. `foreach` doesn't exist with ANSI C++ but is an extension of ANSI C++/CLI. Unlike the C# `foreach`, in C++/CLI there's a blank space between `for` and `each`. F# uses this functionality for the `for/in/do` keywords:

```

// C#
int[] arr = {1, 2, 3};
foreach (int i in arr)
{
    Console.WriteLine(i);
}

// C++/CLI
array<int>^ arr = {1, 2, 3};
for each (int i in arr)
{
    Console::WriteLine(i);
}

' Visual Basic
Dim arr() As Integer = New Integer() {1, 2, 3}
Dim num As Integer
For Each num as Integer In arr
    Console.WriteLine(num)
Next

// F#
for i in arr do
    printfn i

```

foreach makes it easy to iterate through a collection and C# supports creating enumerations by using the `yield` statement. With Visual Basic and C++/CLI, the `yield` statement is not available. Instead, it is necessary to implement the interfaces `IEnumerable` and `IEnumerator` manually with these languages. The `yield` statement is explained in Chapter 6.

EXCEPTION HANDLING

Exception handling is discussed in Chapter 15, “Errors and Exceptions.” This is extremely similar among three languages. All these languages use `try/catch/finally` for handling exceptions, and the `throw` keyword to create an exception. F# uses `raise` for throwing exceptions, and `try/with/finally` for handling exceptions:

 Available for download on Wrox.com

```

// C#
public void Method(Object o)
{
    if (o == null)
        throw new ArgumentException("Error");
}
public void Foo()
{

```

```

    try
    {
        Method(null);
    }
    catch (ArgumentException ex)
    { }
    catch (Exception ex)
    { }
    finally
    { }
}

```

code snippet CSharp/ExceptionDemo.cs



```

// C++/CLI
public:
    void Method(Object^ o)
    {
        if (o == nullptr)
            throw gcnew ArgumentException("Error");
    }
    void Foo()
    {
        try
        {
            Method(nullptr);
        }
        catch (ArgumentException^ ex)
        { }
        catch (Exception^ ex)
        { }
        finally
        { }
    }
}

```

code snippet CPPCLI/ExceptionDemo.h



```

' Visual Basic
Public Sub Method(ByVal o As Object)
    If o = Nothing Then
        Throw New ArgumentException("Error")
    End Sub
Public Sub Foo()
    Try
        Method(Nothing)
    Catch ex As ArgumentException
    ,
    Catch ex As Exception
    ,
    Finally
    ,
    End Try
End Sub

```

code snippet VisualBasic/ExceptionDemo.vb



```

// F#
type ExceptionDemo() as this =
    member this.Method(o : obj) =
        if o = null then
            raise(ArgumentException("error"))
        member this.Foo =
            try
                try

```

```
        this.Method(null)
    with
    | :? ArgumentException as ex -> printfn "%s" ex.Message
    | :? IOException as ex -> printfn "%s" ex.Message
    finally
        printfn "finally"
```

code snippet FSharp/ExceptionDemo.fs

INHERITANCE

.NET languages offer many keywords to define polymorphic behavior, to override or hide methods, access modifiers to allow or not allow member access. For C#, this functionality is discussed in Chapter 4, “Inheritance.” The functionality of C#, C++/CLI, and Visual Basic is very similar, but the keywords are different.

Access Modifiers

The access modifiers of C++/CLI, Visual Basic, and F# are very similar to C#, with some notable differences. Visual Basic uses the `Friend` access modifier instead of `internal` for accessing the types in the same assembly. C++/CLI has one more access modifier: `protected private`. An `internal protected` modifier allows accessing the members from within the same assembly, and also from other assemblies if the type is derived from the base type. C# and Visual Basic don't have a way to allow only derived types within the same assembly. This is possible with `protected private` from C++/CLI. Here `private` means that outside the assembly there's no access, but from inside the assembly `protected` access is possible. The order — whether you write `protected private` or `private protected` — does not matter. The access modifier allowing more is always located within the assembly, and the access modifier allowing less is always outside the assembly. F# uses three access modifiers but also allows the `protected` modifier if the type should be used from other .NET languages. Access modifiers can be declared in F# within signature files.

The following table lists the access modifiers for each of the languages.

C#	C++/CLI	VISUAL BASIC	F#
public	Public	Public	public
protected	Protected	Protected	
private	Private	Private	private
internal	Internal	Friend	internal
internal protected	internal protected	Protected Friend	
not possible	protected private	not possible	

F# signature files have the file extension `.fsi` and define the signature of the members. They can be automatically created from code files with the `--sig` compiler option. According to the requirements, only the access modifiers need to be changed:

```
// F#
type public Person =
    class
        interface IDisplay
            public new : unit -> Person
            public new : firstName:string * lastName:string -> Person
            override ToString : unit -> string
            member public FirstName : string
            member public LastName : string
            member public FirstName : string with set
            member public LastName : string with set
        end
```

Keywords

Keywords important for inheritance are mapped in the following table.

C#	C++/CLI	VISUAL BASIC	F#	FUNCTIONALITY
:	:	Implements	interface with	Implement an interface
:	:	Inherits	inherit	Inherits from a base class
virtual	virtual	Overridable	abstract	Declare a method to support polymorphism
overrides	override	Overrides	override	Override a virtual method
new	new	Shadows		Hide a method from a base class
abstract	abstract	MustInherit	[<AbstractClass>]	Abstract class
sealed	sealed	NotInheritable	[<Sealed>]	Sealed class
abstract	abstract	MustOverride	abstract	Abstract method
sealed	sealed	NotOverridable		Sealed method
this	this	Me		Reference the current object
base	Classname::	MyBase	base	Reference the base class

The order in which you place the keywords is important in the languages. In the code example, an abstract base class `Base` with one abstract method and one implemented method that is virtual are defined. The class `Base` is derived from the class `Derived`, where the abstract method is implemented, and the virtual method is overridden:



```
// C#
public abstract class Base
{
    public virtual void Foo()
    {
    }
    public abstract void Bar();
}
public class Derived: Base
{
    public override void Foo()
    {
        base.Foo();
    }
    public override void Bar()
    {
    }
}
```

code snippet CSharp/InheritanceDemo.cs



```
// C++/CLI
public ref class Base abstract
{
public:
    virtual void Foo()
    {
    }
    virtual void Bar() abstract;
};
public ref class Derived: public Base
{
}
```

```

public:
    virtual void Foo() override
    {
        Base::Foo();
    }
    virtual void Bar() override
    {
    }
};

```

code snippet CPPCLI/InheritanceDemo.h



```

' Visual Basic
Public MustInherit Class Base
    Public Overridable Sub Foo()
    End Sub
    Public MustOverride Sub Bar()
End Class
Public class Derived
    Inherits Base
    Public Overrides Sub Foo()
        MyBase.Foo()
    End Sub
    Public Overrides Sub Bar()
    End Sub
End Class

```

code snippet VisualBasic/InheritanceDemo.vb



```

// F#
[<AbstractClass>]
type Base() as this =
    abstract Foo : unit -> unit
    default this.Foo() = printfn "Base.Foo"
    abstract Bar : unit -> unit

type Derived() as this =
    inherit Base()
    override this.Foo() =
        base.Foo()
        printfn "Derived.Foo"
    override this.Bar() = printfn "Derived.Bar"

```

code snippet FSharp/InheritanceDemo.fs

RESOURCE MANAGEMENT

Working with resources is covered in Chapter 13, “Memory Management and Pointers,” both implementing the `IDisposable` interface and implementing a finalizer. How this looks in C++/CLI, Visual Basic, and F# is covered in this section.

IDisposable Interface Implementation

For freeing resources, the interface `IDisposable` defines the `Dispose()` method. Using C#, Visual Basic, and F# you have to implement the interface `IDisposable`. With C++/CLI the interface `IDisposable` is implemented as well, but this is done by the compiler if you just write a destructor:



```

// C#
public class Resource: IDisposable
{
    public void Dispose()
    {
    }
}

```

```

        // release resource
    }
}

```

code snippet CSharp/Resource.cs



Available for
download on
Wrox.com

```

// C++/CLI
public ref class Resource
{
public:
    ~Resource()
    {
        // release resource
    }
};

```

code snippet CPPCLI/Resource.h



Available for
download on
Wrox.com

```

' Visual Basic
Public Class Resource
    Implements IDisposable
    Public Sub Dispose() Implements IDisposable.Dispose
        ' release resource
    End Sub
End Class

```

code snippet VisualBasic/Resource.vb



Available for
download on
Wrox.com

```

// F#
type Resource() as this =
    interface IDisposable with
        member this.Dispose() = printfn "release resource"

```

code snippet FSharp/Resource.fs



With C++/CLI, the `Dispose()` method is invoked by using the `delete` statement.

using Statement

The C# using statement implements an acquire/use/release pattern to release a resource as soon as it is no longer used, even in the case of an exception. The compiler creates a `try/finally` statement and invokes the `Dispose` method inside the `finally`. Visual Basic supports the `using` statement just as C# does. C++/CLI has an even more elegant approach to this problem. If a reference type is declared locally, the compiler creates a `try/finally` statement to invoke the `Dispose()` method at the end of the block. F# offers two different constructs to easily support resource management. The `use` binding automatically invokes the `Dispose()` method when the value goes out of scope. The `using` expression creates an object that must be disposed and it is disposed at the end of the function that is called. Here, the function `bar` is called after the `Resource` object is created:

```

// C#
using (Resource r = new Resource())
{
    r.Foo();
}

// C++/CLI
{
    Resource r;
    r.Foo();
}

```



```

    }

    ' Visual Basic
    Using r As New Resource
        r.Foo()
    End Using

    // F# use binding
    let rs =
        use r = new Resource()
        r.Foo()
        // r.Dispose called implicitly
    // F# using expression
    let bar (r : Resource) =
        r.Foo()

    using (new Resource()) bar

```

Override Finalize

If a class contains native resources that must be freed, the class must override the `Finalize()` method from the `Object` class. With C#, this is done by writing a destructor. C++/CLI has a special syntax with the `!` prefix to define a finalizer. Within a finalizer, you cannot dispose contained objects that have a finalizer as well because the order of finalization is not guaranteed. That's why the `Dispose` pattern defines an additional `Dispose()` method with a Boolean parameter. With C++/CLI, it is not necessary to implement this pattern in the code because this is done by the compiler. The C++/CLI destructor implements both `Dispose()` methods. With Visual Basic, both `Dispose()` and the finalizer must be implemented manually. However, most Visual Basic classes do not use native resources directly, just with the help of other classes. With Visual Basic, usually it is not necessary to override the `Finalize()` method, but an implementation of the `Dispose()` method is often required.



Writing a destructor with C# overrides the `Finalize()` method of the base class. A C++/CLI destructor implements the `IDisposable` interface.



```

// C#
public class Resource: IDisposable
{
    ~Resource // override Finalize
    {
        Dispose(false);
    }
    protected virtual void Dispose(bool disposing)
    {
        if (disposing) // dispose embedded members
        {
        }
        // release resources of this class
        GC.SuppressFinalize(this);
    }
    public void Dispose()
    {
        Dispose(true);
    }
}

```

code snippet CSharp/Resource.cs



Available for
download on
Wrox.com

```
// C++/CLI
public ref class Resource
{
public:
    ~Resource() // implement IDisposable
    {
        this->!Resource();
    }
    !Resource() // override Finalize
    {
        // release resource
    }
};
```

code snippet CPPCLI/Resource.h



Available for
download on
Wrox.com

```
' Visual Basic
Public Class Resource
    Implements IDisposable
    Public Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
    Protected Overridable Sub Dispose(ByVal disposing)
        If disposing Then
            ' Release embedded resources
        End If
        ' Release resources of this class
    End Sub
    Protected Overrides Sub Finalize()
        Try
            Dispose(False)
        Finally
            MyBase.Finalize()
        End Try
    End Sub
End Class
```

code snippet VisualBasic/Resource.vb

DELEGATES

Delegates — type-safe pointers to methods — are discussed in Chapter 8, “Delegates, Lambdas, and Events.” In all four languages, the keyword `delegate` can be used to define a delegate. The difference is with using the delegate.

The sample code shows a class `Demo` with a static method `Foo()` and an instance method `Bar()`. Both of these methods are invoked by delegate instances of type `DemoDelegate`. `DemoDelegate` is declared to invoke a method with `void` return type and an `int` parameter.

When using the delegate, C# supports delegate inference, where the compiler creates a delegate instance and passes the address of the method.

With C# and C++/CLI, two delegates can be combined into one by using the `+` operator:



Available for
download on
Wrox.com

```
// C#
public delegate void DemoDelegate(int x);
public class Demo
{
    public static void Foo(int x) { }
    public void Bar(int x) { }
}
```

```

Demo d = new Demo();
DemoDelegate d1 = Demo.Foo;
DemoDelegate d2 = d.Bar;
DemoDelegate d3 = d1 + d2;
d3(11);

```

code snippet CSharp/DelegateSample.cs

Delegate inference is not possible with C++/CLI. With C++/CLI, you must create a new instance of the delegate type and pass the address of the method to the constructor:



```

// C++/CLI
public delegate void DemoDelegate(int x);
public ref class Demo
{
public:
    static void Foo(int x) { }
    void Bar(int x) { }
};
Demo^ d = gcnew Demo();
DemoDelegate^ d1 = gcnew DemoDelegate(&Demo::Foo);
DemoDelegate^ d2 = gcnew DemoDelegate(d, &Demo::Bar);
DemoDelegate^ d3 = d1 + d2;
d3(11);

```

code snippet CPPCLI/DelegateSample.h

Similarly to C++/CLI, Visual Basic does not support delegate inference. You have to create a new instance of the delegate type and pass the address of a method. Visual Basic has the `AddressOf` operator to pass the address of a method.

Visual Basic doesn't overload the `+` operator for delegates, so it is necessary to invoke the `Combine()` method from the `Delegate` class. The `Delegate` class is written inside brackets because `Delegate` is a Visual Basic keyword, and thus it is not possible to use a class with the same name. Putting brackets around `Delegate` ensures that the class is used instead of the `Delegate` keyword:



```

' Visual Basic
Public Delegate Sub DemoDelegate(ByVal x As Integer)
Public Class Demo
    Public Shared Sub Foo(ByVal x As Integer)
    ,
    End Sub
    Public Sub Bar(ByVal x As Integer)
    ,
    End Sub
End Class
Dim d As New Demo()
Dim d1 As New DemoDelegate(AddressOf Demo.Foo)
Dim d2 As New DemoDelegate(AddressOf d.Bar)
Dim d3 As DemoDelegate = [Delegate].Combine(d1, d2)
d3(11)

```

code snippet VisualBasic/DelegateDemo.vb

Functions as objects are first-class citizens with F#. Just as with interoperability with other .NET languages, delegates are needed — therefore the syntax seems more complex. With F#, a delegate is declared assigning the delegate keyword to the type name and defining the parameters and return type. With `DemoDelegate`, the parameter is of type `int` and the return type `unit` which is similar to `void` in C#.

The `Demo` type is defined with a static member `Foo` and an instance member `Bar`. Both of these members fulfill the requirements of the delegate type.

The variable `d1` is a delegate variable that references the `Demo.Foo` method, and `d2` references the instance method `Bar`. The function `invokeDelegate`, with parameters `DemoDelegate` and `int`, is declared to invoke the functions referenced by the delegate. It calls the `Invoke` method of the `Delegate` class and passes the `int` parameter. After the `invokeDelegate` function is declared, it is invoked by passing the delegate instance `d1` and the value `33`. To combine delegates, you need to invoke the `Delegate.Combine()` method. Because `Combine()` requires two `Delegate` parameters and returns a `Delegate` type, upcasting and downcasting is required. Here, the F# syntax `:>` is used to cast `d1` to the base type `Delegate`, and `:?>` to cast the `Delegate` type `d33` to the derived class `DemoDelegate`. Instead of using `:>` and `:?>` it is also legal to use the keywords `upcast` and `downcast`:



```
// F#
type DemoDelegate = delegate of int -> unit

type Demo() as this =
    static member Foo(x : int) =
        printfn "Foo %d" x
    member this.Bar(x : int) =
        printfn "Bar %d" x

// F# using delegate
let d = Demo()
let d1 : DemoDelegate = new DemoDelegate(Demo.Foo)
let invokeDelegate (dlg : DemoDelegate) (x : int) =
    dlg.Invoke(x)
(involveDelegate d1 33)
let d2 : DemoDelegate = new DemoDelegate(d.Bar)
let d11 = d1 :> Delegate
let d22 = d2 :> Delegate
let d33 : Delegate = Delegate.Combine(d11, d22)
let d3 : d33 :?> DemoDelegate
(involveDelegate d3 11)
```

code snippet FSharp/DelegateDemo.fs

EVENTS

With the `event` keyword, a subscription mechanism can be done that is based on delegates. Again, all languages define an `event` keyword for offering events from a class. The class `EventDemo` fires events with the name `DemoEvent` of type `DemoDelegate`.

In C#, the syntax for firing the event looks similar to a method call of the event. The event variable is `null` as long as nobody is registered to the event, so a check for not `null` must be done before firing the event. The handler method is registered by using the `+=` operator and passing the address of the handler method with the help of delegate inference:



```
// C#
public class EventDemo
{
    public event DemoDelegate DemoEvent;
    public void FireEvent()
    {
        if (DemoEvent != null)
            DemoEvent(44);
    }
}

public class Subscriber
{
    public void Handler(int x)
    {

```

```

        // handler implementation
    }
}
//...
EventDemo evd = new EventDemo();
Subscriber subscr = new Subscriber();
evd.DemoEvent += subscr.Handler;
evd.FireEvent();

```

code snippet CSharp/EventDemo.cs

C++/CLI is very similar to C# except that when firing the event, you do not need to first see that the event variable is not null. This is automatically done by the IL code created from the compiler.



Both C# and C++/CLI use the += operator to unregister from an event.



```

// C++/CLI
public ref class EventDemo
{
public:
    event DemoDelegate^ DemoEvent;
    public void FireEvent()
    {
        DemoEvent(44);
    }
}
public class Subscriber
{
public:
    void Handler(int x)
    {
        // handler implementation
    }
}
//...
EventDemo^ evd = gcnew EventDemo();
Subscriber^ subscr = gcnew Subscriber();
evd->DemoEvent += gcnew DemoDelegate(subscr, &Subscriber::Handler);
evd->FireEvent();

```

code snippet CPPCLI/EventDemo.h

Visual Basic has a different syntax. The event is declared with the `Event` keyword, which is the same as in C# and C++/CLI. However, the event is raised with the `RaiseEvent` statement. The `RaiseEvent` statement checks whether the event variable is initialized by a subscriber. To register a handler, the `AddHandler` statement has the same functionality as the `+=` operator in C#. `AddHandler` requires two parameters: the first defines the event, the second the address of the handler. The `RemoveHandler` statement is used to unregister a handler from the event:



```

' Visual Basic
Public Class EventDemo
    Public Event DemoEvent As DemoDelegate
    public Sub FireEvent()
        RaiseEvent DemoEvent(44);
    End Sub
End Class
Public Class Subscriber
    Public Sub Handler(ByVal x As Integer)
        ' handler implementation
    End Sub
End Class

```

```

        End Sub
    End Class
'...
Dim evd As New EventDemo()
Dim subscr As New Subscriber()
AddHandler evd.DemoEvent, AddressOf subscr.Handler
evd.FireEvent()

```

code snippet VisualBasic/EventDemo.vb

Visual Basic offers another syntax that is not available with the other languages: you can also use the `Handles` keyword with the method that subscribes to the event. The requirement for this is to define a variable with the `WithEvents` keyword:

```

Public Class Subscriber
    Public WithEvents evd As EventDemo
    Public Sub Handler(ByVal x As Integer) Handles evd.DemoEvent
        ' Handler implementation
    End Sub
    Public Sub Action()
        evd = New EventDemo()
        evd.FireEvent()
    End Sub
End Class

```

GENERICIS

All four languages support the creation and use of generics. Generics are discussed in Chapter 5, “Generics.”

To use generics, C# borrowed the syntax from C++ templates to define the generic type with angle brackets. C++/CLI uses the same syntax. In Visual Basic, the generic type is defined with the `Of` keyword in braces. Here, F# looks very similar to C#:



Available for
download on
Wrox.com

```

// C#
List<int> intList = new List<int>();
intList.Add(1);
intList.Add(2);
intList.Add(3);

```

code snippet CSharp/GenericsDemo.cs



Available for
download on
Wrox.com

```

// C++/CLI
List<int>^ intList = gcnew List<int>();
intList->Add(1);
intList->Add(2);
intList->Add(3);

```

code snippet CPPCLI/GenericsDemo.h



Available for
download on
Wrox.com

```

' Visual Basic
Dim intList As List(Of Integer) = New List(Of Integer)()
intList.Add(1)
intList.Add(2)
intList.Add(3)

```

code snippet VisualBasic/GenericsDemo.vb



Available for
download on
Wrox.com

```

// F#
let intList =
    new List<int>()
intList.Add(1)

```

```
intList.Add(2)
intList.Add(3)
```

code snippet FSharp/GenericsDemo.fs

Because you use angle brackets with the class declaration, the compiler knows to create a generic type. Constraints are defined with the `where` clause:



Available for
download on
Wrox.com

```
public class MyGeneric<T>
    where T: IComparable<T>
{
    private List<T> list = new List<T>();
    public void Add(T item)
    {
        list.Add(item);
    }
    public void Sort()
    {
        list.Sort();
    }
}
```

code snippet CSharp/GenericsDemo.cs

Defining a generic type with C++/CLI is similar to defining a template with C++. Instead of the `template` keyword, with generics the `generic` keyword is used. The `where` clause is similar to that in C#; however, C++/CLI does not support a constructor constraint:



Available for
download on
Wrox.com

```
generic <typename T>
where T: IComparable<T>
ref class MyGeneric
{
private:
    List<T>^ list;
public:
    MyGeneric()
    {
        list = gcnew List<T>();
    }
    void Add(T item)
    {
        list->Add(item);
    }
    void Sort()
    {
        list->Sort();
    }
};
```

code snippet CPPCLI/GenericsDemo.h

Visual Basic defines a generic class with the `Of` keyword. Constraints can be defined with `As`:



Available for
download on
Wrox.com

```
Public Class MyGeneric(Of T As IComparable(Of T))
    Private myList = New List(Of T)
    Public Sub Add(ByVal item As T)
        myList.Add(item)
    End Sub
    Public Sub Sort()
        myList.Sort()
    End Sub
End Class
```

code snippet VisualBasic/GenericsDemo.vb

F# defines a generic type with angle brackets. The generic type is marked with an apostrophe. Constraints are defined with the `when` keyword. A constraint follows `when`. In the example, a type constraint with `:>` is used. This defines that the generic type must derive from the interface or base class. You can also define value type constraints (`: struct`), reference type constraints (`: not struct`), and default constructor constraints (`'a : (new : unit -> 'a)`). A constraint that is not available with other languages is the null constraint (`'a : null`). This constraint defines that the `'a` generic type needs to be nullable. This allows all .NET reference types, but not F# types that are not nullable:



Available for
download on
Wrox.com

```
type MyGeneric<'a> when 'a :> IComparable<'a>() as this =
    let list = new List<'a>()
    member this.Add(item : 'a) = list.Add(item)
    member this.Sort() = list.Sort()
```

code snippet FSharp/GenericsDemo.fs

LINQ QUERIES

Language-integrated queries are a feature of C# and Visual Basic. The syntax is very similar between these two languages:



LINQ is discussed in Chapter 11, “Language Integrated Query.”



Available for
download on
Wrox.com

```
// C#
var query = from r in racers
             where r.Country == "Brazil"
             orderby r.Wins descending
             select r;
```

code snippet CSharp/LinqSample.cs



Available for
download on
Wrox.com

```
' Visual Basic
Dim query = From r in racers _
             Where r.Country = "Brazil" _
             Order By r.Wins Descending _
             Select r
```

code snippet VisualBasic/LinqSample.vb



C++/CLI does not support LINQ queries.

C++/CLI MIXING NATIVE AND MANAGED CODE

One of the big advantages of C++/CLI is the capability to mix native and managed code. You use native code from C# through a mechanism known as *platform invoke*, which is discussed in Chapter 26, “Interop.” Using native code from C++/CLI is known as *It just works*.

In a managed class, you can use both native and managed code, as you can see here. The same is true for a native class. You can mix native and managed code as well within a method:



Available for
download on
Wrox.com

```
#pragma once
#include <iostream>           // include this header file for cout
using namespace std;        // the iostream header defines the namespace std
using namespace System;
```



```

public ref class Managed
{
public:
    void MixNativeAndManaged()
    {
        cout << "Native Code" << endl;
        Console::WriteLine("Managed Code");
    }
};

```

code snippet CPPCLI/Mixing.h

In a managed class, you can also declare a field of a native type or a pointer to a native type. Doing the same the other way around is not possible to accomplish directly. You must take care that an instance of a managed type can be moved by the garbage collector when cleaning up memory.

To use managed classes as a member within native classes, C++/CLI defines the keyword `gcroot`, which is defined in the header file `gcroot.h`. The `gcroot` keyword wraps a `GCHandle` that keeps track of a CLR object from a native reference:

```

#pragma once
#include "gcroot.h"
using namespace System;
public ref class Managed
{
public:
    Managed() { }
    void Foo()
    {
        Console::WriteLine("Foo");
    }
};
public class Native
{
private:
    gcroot<Managed^> m_p;
public:
    Native()
    {
        m_p = gcnew Managed();
    }
    void Foo()
    {
        m_p->Foo();
    }
};

```

C# SPECIFICS

Some C# syntax features were not covered in this chapter. C# defines the `yield` statement, which makes it easy to create enumerators. This statement is not available with C++/CLI and Visual Basic; with these languages an enumerator must be implemented manually. Also, C# defines a special syntax for nullable types, whereas with the other languages you have to use the generic struct `Nullable<T>` instead.

C# allows for unsafe code blocks where you can use pointers and pointer arithmetic. This feature can be extremely helpful for invoking methods from native libraries. Visual Basic does not have this capability; this is a real advantage of C#. C++/CLI does not need the `unsafe` keyword to define unsafe code blocks. It's very natural with C++/CLI to mix native and managed code.

SUMMARY

In this chapter, you learned how to map the syntax from C# to Visual Basic, C++/CLI, and F#. C++/CLI defines extensions to C++ for writing .NET applications and draws on C# for the syntax extensions. Although C# and C++/CLI have the same roots, there are many important differences. Visual Basic does not use curly brackets, but is chattier instead. The syntax from F# is very different as its major focus is on functional programming.

With the syntax mapping, you've seen how to map the C# syntax to C++/CLI, Visual Basic, and F#; how the other three languages look, with defining types, methods, and properties; what keywords are used for OO features; how resource management is done; and how delegates, events, and generics are implemented with the four languages.

Although it is possible to map most of the syntax, the languages are still different in their functionality.